Kinds Are Calling Conventions

Paul Downen, Zena M. Ariola,
Simon Peyton Jones, Richard A. Eisenberg

PLDI 2022, June 15—17
Originally in ICFP 2020, August 23—29

Efficient Function Calls

Parameter Passing Techniques

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- Arity — How many?

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- Arity — How many?
- Levity (aka Evaluation Strategy) — When to compute?

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- Arity — How many?
- Levity (aka Evaluation Strategy) — When to compute?

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
     let z = expensive x
     in y + z
```

Arity 2

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
      let z = expensive x
      in y + z
```

Arity 2

```
f2 = \x -> f1 x
```

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
      let z = expensive x
      in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
      let z = expensive x
      in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Arity 1

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

```
f1, f2, f3, f4 :: Int -> Int -> Int
```

Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Arity 1

```
f4 = \x -> f3 x
```

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int` Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Arity 1

```
f4 = \x -> f3 x
   ≠ \x -> \y -> f3 x y
```

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int` Type suggests arity 2

```
f1 = \x -> \y ->
       let z = expensive x
       in y + z
```

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

Arity 2

```
f3 = \x ->
       let z = expensive x
       in \y -> y + z
```

Arity 1

```
f4 = \x -> f3 x
   ≠ \x -> \y -> f3 x y
```

Arity 1

Hint: ‘`expensive x`’ may be costly, or even cause side effects

What Is Arity?

For Curried Functions

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do "serious work."

- If '`f 1 2 3`' does work, but '`f 1 2`' does not, then '`f`' has arity 3

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do "serious work."

- If '`f 1 2 3`' does work, but '`f 1 2`' does not, then '`f`' has arity 3

Definition 2. The number of times a function may be η-expanded without changing its behavior or cost.

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘`f 1 2 3`’ does work, but ‘`f 1 2`’ does not, then ‘`f`’ has arity 3

Definition 2. The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘`f`’ is equivalent to ‘`\x y z -> f x y z`’, then ‘`f`’ has arity 3

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do "serious work."

- If '`f 1 2 3`' does work, but '`f 1 2`' does not, then '`f`' has arity 3

Definition 2. The number of times a function may be η-expanded without changing its behavior or cost.

- If '`f`' is equivalent to '`\x y z -> f x y z`', then '`f`' has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘`f 1 2 3`’ does work, but ‘`f 1 2`’ does not, then ‘`f`’ has arity 3

Definition 2. The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘`f`’ is equivalent to ‘`\x y z -> f x y z`’, then ‘`f`’ has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

- If ‘`f`’ has arity 3, then ‘`f 1 2 3`’ can be implemented as a single call

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘`f 1 2 3`’ does work, but ‘`f 1 2`’ does not, then ‘`f`’ has arity 3

Definition 2. The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘`f`’ is equivalent to ‘`\x y z -> f x y z`’, then ‘`f`’ has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

- If ‘`f`’ has arity 3, then ‘`f 1 2 3`’ can be implemented as a single call

Goal: An IL with ***un**restricted* η for functions, along with *restricted* β for other types

Static Arity

In an Intermediate Language

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘`a ~> b`’)
  - To distinguish from the source-level $a \to b$ with different semantics

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII '`a ~> b`')
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like source functions
  - $\lambda x\,.f\ x =_\eta f : a \rightsquigarrow b$ *unconditionally*

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII '`a ~> b`')
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like source functions
  - $\lambda x\,.f\ x =_\eta f : a \rightsquigarrow b$ *unconditionally*
- Application may still be *restricted* for efficiency, like source functions
  - $(\lambda x\,.\,x + x)\ (\mathit{expensive}\ \ 10^6)$ does not recompute $\mathit{expensive}\ \ 10^6$

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘`a ~> b`’)
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like source functions
  - $\lambda x \,.\, f\ x =_\eta f : a \rightsquigarrow b$ *unconditionally*
- Application may still be *restricted* for efficiency, like source functions
  - $(\lambda x \,.\, x + x)\ (\mathit{expensive}\ \ 10^6)$ does not recompute $\mathit{expensive}\ \ 10^6$
- With full η, types express arity — just count the arrows
  - $f : \mathit{Int} \rightsquigarrow \mathit{Bool} \rightsquigarrow \mathit{String}$ has arity 2, no matter f’s definition

Currying

When Partial Application Matters

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η, f3 now has arity 2, not 1!

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

```
f3' :: Int ~> { Int ~> Int }
f3' = \x -> let z = expensive x in Clos (\y -> y + z)
```

```
Clos :: (Int ~> Int) ~> {Int ~> Int}
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

```
f3' :: Int ~> { Int ~> Int }
f3' = \x -> let z = expensive x in Clos (\y -> y + z)
```

- `f3'` is an arity 1 function; returns a closure `{Int~>Int}` of an arity 1 function

```
Clos :: (Int ~> Int) ~> {Int ~> Int}
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

```
f3' :: Int ~> { Int ~> Int }
f3' = \x -> let z = expensive x in Clos (\y -> y + z)
```

- f3' is an arity 1 function; returns a closure {Int~>Int} of an arity 1 function
  - map (App (f3' 100)) [1..10^6] computes 'expensive 100' only once ☺

```
Clos :: (Int ~> Int) ~> {Int ~> Int}   App  :: {Int ~> Int} ~> Int ~> Int
```

Functions are *Called*

Not *Evaluated*

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded
  - Call-by-name: later, re-evaluated every time `f` is demanded

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded
  - Call-by-name: later, re-evaluated every time `f` is demanded

```
x' = let f :: Int ~> Int = \y -> expensive 100 y in …f…f…
```

Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded
  - Call-by-name: later, re-evaluated every time `f` is demanded

```
x' = let f :: Int ~> Int = \y -> expensive 100 y in …f…f…
```

- `x = x'` by η, and `x'` always follows call-by-name order!

Functions are *Called*

Not *Evaluated*

`x = let f :: Int ~> Int = expensive 100 in …f…f…`

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded
  - Call-by-name: later, re-evaluated every time `f` is demanded

`x’ = let f :: Int ~> Int = \y -> expensive 100 y in …f…f…`

- `x = x’` by η, and `x’` always follows call-by-name order!
- Primitive functions are never just *evaluated*; they are always *called*

The Problem With Polymorphism

And Static Compilation

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows…

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~> a` has arity 2

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~> a` has arity 2
  - `g :: Int ~> a` has arity 1

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~> a` has arity 2
  - `g :: Int ~> a` has arity 1
- But what if a = Bool ~> Bool?

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~>` *a* has arity 2
  - `g :: Int ~>` *a* has arity 1
- But what if a = Bool ~> Bool?
  - `f :: Int ~> Int ~> Bool ~> Bool` has arity 3...

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~> a` has arity 2
  - `g :: Int ~> a` has arity 1
- But what if a = Bool ~> Bool?
  - `f :: Int ~> Int ~> Bool ~> Bool` has arity 3...
  - `g :: Int ~> Bool ~> Bool` has arity 2... oops...

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
  - `f :: Int ~> Int ~> a` has arity 2
  - `g :: Int ~> a` has arity 1
- But what if a = Bool ~> Bool?
  - `f :: Int ~> Int ~> Bool ~> Bool` has arity 3...
  - `g :: Int ~> Bool ~> Bool` has arity 2... oops...
- How to statically compile? Is ‘`g 5`’ a call? A partial application?

Nonuniform Representation

And Static Compilation

Nonuniform Representation

And Static Compilation

- Primitive types:

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

- Different sizes

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

- Different sizes
- Different locations

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
- Different levity

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

- Different sizes
- Different locations
- Different levity

```
revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x
```

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

- Different sizes
- Different locations
- Different levity

```
revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x

(++)         :: [a]    -> [a]    -> [a]
plusFloat# :: Float# -> Float# -> Float#
```

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

- Different sizes
- Different locations
- Different levity

```
revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x
```

```
(++)        :: [a]    -> [a]    -> [a]
plusFloat# :: Float# -> Float# -> Float#
```

`revapp [0..3] (++ [4..9])` vs `revapp 2.5 (plusFloat# 1.5)`

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - ‘*a*’ is represented as a pointer

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - ‘*a*’ is represented as a pointer
  - ‘*a*’ has arity 0

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - ‘*a*’ is represented as a pointer
  - ‘*a*’ has arity 0
- Restriction on quantifiers `forall a::k. …`

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - '*a*' is represented as a pointer
  - '*a*' has arity 0
- Restriction on quantifiers `forall a::k.` …
  - Special kinds for unboxed (#) and non-zero arity (~) types

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - '*a*' is represented as a pointer
  - '*a*' has arity 0
- Restriction on quantifiers `forall a::k.` …
  - Special kinds for unboxed (#) and non-zero arity (~) types
  - k may be ⋆ or ⋆->⋆ but never # or ~

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - '*a*' is represented as a pointer
  - '*a*' has arity 0
- Restriction on quantifiers `forall a::k.` …
  - Special kinds for unboxed (#) and non-zero arity (~) types
  - k may be ⋆ or ⋆->⋆ but never # or ~
- Draconian restriction is unsatisfactory

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - '*a*' is represented as a pointer
  - '*a*' has arity 0
- Restriction on quantifiers `forall a::k. …`
  - Special kinds for unboxed (#) and non-zero arity (~) types
  - k may be ⋆ or ⋆->⋆ but never # or ~
- Draconian restriction is unsatisfactory
  - **Too restrictive**: Identical definitions/code repeated for different types (like `error :: String -> a`)

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - '*a*' is represented as a pointer
  - '*a*' has arity 0
- Restriction on quantifiers `forall a::k.` …
  - Special kinds for unboxed (#) and non-zero arity (~) types
  - k may be ⋆ or ⋆->⋆ but never # or ~
- Draconian restriction is unsatisfactory
  - **Too restrictive**: Identical definitions/code repeated for different types (like `error :: String -> a`)
  - **Incompatible with kind polymorphism**: `forall k::Kind. forall a::k.  ???`

Representation Polymorphism

Kinds As Representations

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \mathrm{TYPE}\ r$

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \mathrm{TYPE}\ r$
  - $r :: \mathrm{Rep}$ is the *representation* of a

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of a
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::★` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `★ = TYPE Ptr`

```
revapp x f = f x
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b

revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).
          a -> (a -> b) -> b
```

Representation Polymorphism

Kinds As Representations

- Generalize `a::⋆` to `a::TYPE r`
  - `r::Rep` is the *representation* of `a`
  - `⋆ = TYPE Ptr`

```
revapp x f = f x
```

```
revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
          a -> (a -> b) -> b
```

```
revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).
          a -> (a -> b) -> b
```

Arity Polymorphism

Kinds As Calling Conventions

Arity Polymorphism

- Generalize `a::TYPE r` to `a::TYPE r c`

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE  r` to `a::TYPE  r  c`
  - `r::Rep` is the *runtime representation* of a

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of `a`
  - `c::Conv` is the *calling convention* of `a`

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of `a`
  - `a::TYPE Ptr Call[n]` says values of `a` are pointers with arity n (simplified)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

- `f :: Int ~> Int ~> a :: TYPE Ptr Call[4]` has arity 4 (2 + 1 + 1)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

  - `f :: Int ~> Int ~> a :: TYPE Ptr Call[4]` has arity 4 (2 + 1 + 1)
  - `g :: Int ~> a :: TYPE PTR Call[3]` has arity 3 (2 + 1)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

  - `f :: Int ~> Int ~> a :: TYPE Ptr Call[4]` has arity 4 (2 + 1 + 1)
  - `g :: Int ~> a :: TYPE PTR Call[3]` has arity 3 (2 + 1)

```
revapp :: forall (c::Conv) (r::Rep)
          (a::TYPE Ptr c) (b::TYPE r Call[1]).
          a ~> (a ~> b) ~> b
revapp x f = f x
```

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

- `f :: Int ~> Int ~> a :: TYPE Ptr Call[4]` has arity 4 (2 + 1 + 1)
- `g :: Int ~> a :: TYPE PTR Call[3]` has arity 3 (2 + 1)

```
revapp :: forall (c::Conv) (r::Rep)
          (a::TYPE Ptr c) (b::TYPE r Call[1]).
          a ~> (a ~> b) ~> b
revapp x f = f x
```

- `f :: a ~> b :: TYPE Ptr Call[2]` has arity 2

Arity Polymorphism

Kinds As Calling Conventions

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of a
  - `c::Conv` is the *calling convention* of a
  - `a::TYPE Ptr Call[n]` says values of a are pointers with arity n (simplified)

```
poly :: forall a::TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

  - `f :: Int ~> Int ~> a :: TYPE Ptr Call[4]` has arity 4 (2 + 1 + 1)
  - `g :: Int ~> a :: TYPE PTR Call[3]` has arity 3 (2 + 1)

```
revapp :: forall (c::Conv) (r::Rep)
          (a::TYPE Ptr c) (b::TYPE r Call[1]).
          a ~> (a ~> b) ~> b
revapp x f = f x
```

  - `f :: a ~> b :: TYPE Ptr Call[2]` has arity 2
  - `x :: a :: TYPE Ptr c` is represented as a pointer

Levity Polymorphism

Call vs Eval, Revisited

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable `g`

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable *g*

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []         = 0
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []       = 0
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []       = 0
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []       = 0
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []         = 0
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
  - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
  - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum []       = 0
sum (x : xs) = x + sum xs
```

✔

Static Compilation

To the Machine

Static Compilation

- Only basic types (pointer, integer, float); no polymorphism

Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
          in (g 4, g 5)
```

Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

η

Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
          in (g 4, g 5)
```

$\Downarrow \eta$

```
poly = \(f::Ptr) ->
```

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

η

```
poly = \(f::Ptr) ->
         let g::Ptr = \(x::I32, y::?, z::?) -> f(3, x, y, z)
```

Static Compilation

With Polymorphic η-Expansion

Static Compilation

With Polymorphic η-Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

$$\Big\Downarrow \eta$$

Static Compilation

With Polymorphic η-Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

η

```
poly = \(f::Ptr) ->
```

Static Compilation

With Polymorphic η-Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

η

```
poly = \(f::Ptr) ->
          let g::Ptr = \(x::I32, y::Ptr, z::F64) -> f(3,x,y,z)
```

Static Compilation

With Polymorphic η-Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

η

```
poly = \(f::Ptr) ->

       let g::Ptr = \(x::I32, y::Ptr, z::F64) -> f(3,x,y,z)

       in (\(y::Ptr, z::F64) -> g(4, y, z),
           \(y::Ptr, z::F64) -> g(5, y, z))
```

Even More

In the Paper

- Levity Polymorphism
  - For when evaluation strategy doesn't matter
- Compiling Source → Intermediate → Target
  - Via kind-directed η-expansion and register assignment
- Type system for ensuring static compilation
  - Of definitions with arity, levity, and representation polymorphism

Kinds capture the details of efficient calling conventions in low-level machine code